Java 8 Programmer II Study Guide

Chapter TWENTY-FIVE
Files and Streams

Exam Objectives

Use Stream API with NIO.2.

New Stream Methods in NIO.2

With the arrival of streams to Java, some difficult file operations to implement in NIO.2 (that often required an entire class) have been simplified.

`java.nio.file.Files` , a class with static methods that we reviewed in the last chapter, added operations that return implementations of the `Stream` interface.

The methods are:

```
static Stream<Path> find(Path start,
            int maxDepth,
            BiPredicate<Path,
                BasicFileAttributes> matcher,
            FileVisitOption... options)

static Stream<Path> list(Path dir)

static Stream<String> lines(Path path)
static Stream<String> lines(Path path, Charset cs)

static Stream<Path> walk(Path start,
                         FileVisitOption... options)
static Stream<Path> walk(Path start,
                         int maxDepth,
                         FileVisitOption... options)
```

An important thing to notice is that the returned streams are **LAZY**, which means that the elements are not loaded (or read) until they are used. This is a great performance enhancement.

Let's describe each method starting with `File.list()` .

Files.list()

```
static Stream<Path> list(Path dir)
            throws IOException
```

This method iterates over a directory to return a stream whose elements are `Path` objects that represent the entries of that directory.

```
try(Stream<Path> stream =
        Files.list(Paths.get("/temp"))) {
    stream.forEach(System.out::println);
} catch(IOException e) {
    e.printStackTrace();
}
```

A possible output:

```
/temp/dir1
/temp/dir2
/temp/file.txt
```

The use of a `try-with-resources` is recommended so the stream's close method can be invoked to close the file system resources.

As you can see, this method lists directories and files in the specified directory. However, it is not recursive, in other words, it **DOESN'T** traverse subdirectories.

Another two important considerations about this method:

- If the argument doesn't represent a directory, an exception is thrown.
- This method is thread safe but is weakly consistent, meaning that while iterating a directory, updates to it may or may not be reflected in the returned stream.

Files.walk()

```
static Stream<Path> walk(Path start,
                         FileVisitOption... options)
            throws IOException
static Stream<Path> walk(Path start,
                         int maxDepth,
                         FileVisitOption... options)
            throws IOException
```

This method also iterates over a directory to return a stream whose elements are `Path` objects that represent the entries of that directory.

The difference with `Files.list()` is that `Files.walk()` **DOES** recursively traverse the subdirectories.

It does it with a *depth-first* strategy, which traverses the directory structure from the root to all the way down a subdirectory before exploring another one.

For example, considering this structure:

```
/temp/
   /dir1/
      /subdir1/
         111.txt
      /subdir2/
         121.txt
         122.txt
   /dir2/
      21.txt
      22.txt
   file.txt
```

The following code:

```
try(Stream<Path> stream =
           Files.walk(Paths.get("c:/temp"))) {
    stream.forEach(System.out::println);
} catch(IOException e) {
    e.printStackTrace();
}
```

Will output:

```
/temp
/temp/dir1
/temp/dir1/subdir1
/temp/dir1/subdir1/111.txt
/temp/dir1/subdir2
/temp/dir1/subdir2/121.txt
/temp/dir1/subdir2/122.txt
/temp/dir2
/temp/dir2/21.txt
/temp/dir2/22.txt
/temp/file.txt
```

By default, this method uses a maximum subdirectory depth of `Integer.MAX_VALUE` . But you can use the overloaded version that takes the maximum depth as the second parameter:

```
try(Stream<Path> stream =
        Files.walk(Paths.get("/temp"), 1)) {
    stream.forEach(System.out::println);
} catch(IOException e) {
    e.printStackTrace();
}
```

The output is:

```
/temp
/temp/dir1
/temp/dir2
/temp/file.txt
```

A value of `0` means that only the starting directory is visited.

Also, this method doesn't follow symbolic links by default.

Symbolic links can cause a *cycle*, an infinite circular dependency between directories. However, this method is smart enough to detect a cycle and throw a `FileSystemLoopException` .

To follow symbolic links, just use the argument of type `FileVisitOption` (preferably, also using the maximum depth argument) this way:

```
try(Stream<Path> stream =
        Files.walk(Paths.get("/temp"),
                   2,
                   FileVisitOption.FOLLOW_LINKS)
) {
    stream.forEach(System.out::println);
} catch(IOException e) {
    e.printStackTrace();
}
```

Just like `Files.list()` , it's recommended to use `Files.walk()` with `try-with-resources` , if the argument doesn't represent a directory, an exception is thrown and it's considered a *weakly consistent* method.

Files.find()

```
static Stream<Path> find(Path start,
        int maxDepth,
        BiPredicate<Path,BasicFileAttributes> matcher,
        FileVisitOption... options)
    throws IOException
```

This method is similar to `Files.walk()` , but takes an additional argument of type `BiPredicate` that is used to filter the files and directories.

Remember that a `BiPredicate` takes two arguments and returns a `boolean` . In this case:

- The first argument is the `Path` object that represents the file or directory.
- The second argument is a `BasicFileAttributes` object that represents the attributes of the file or directory in the file system (like creation time, if it's a file, directory or symbolic link, size, etc.).
- The returned `boolean` indicates if the file should be included in the returned stream.

The following example returns a stream that includes just directories:

```
BiPredicate<Path, BasicFileAttributes> predicate =
    (path, attrs) -> {
        return attrs.isDirectory();
};
int maxDepth = 2;
try(Stream<Path> stream =
        Files.find(Paths.get("/temp"),
                    maxDepth, predicate)) {
    stream.forEach(System.out::println);
} catch(IOException e) {
    e.printStackTrace();
}
```

A possible output can be:

```
/temp
/temp/dir1
/temp/dir1/subdir1
/temp/dir1/subdir2
/temp/dir2
```

Like `Files.walk()` , it can also take a `FileVisitOption` for visiting symbolic links, it's recommended to use it in a `try-with-resources` and throws an exception if it cannot read a file or directory.

Files.lines()

```
static Stream<String> lines(Path path, Charset cs)
        throws IOException
static Stream<String> lines(Path path)
        throws IOException
```

This method reads all the lines of a file as a stream of `String` s.

As the stream is lazy, it doesn't load all the lines into memory, only the line read at any given time. If the file doesn't exist, an exception is thrown.

The file's bytes are decoded using the specified charset or with `UTF-8` by default.

For example:

```
try(Stream<String> stream =
            Files.lines(Paths.get("/temp/file.txt"))) {
        stream.forEach(System.out::println);
} catch(IOException e) {
    e.printStackTrace();
}
```

In Java 8, a `lines()` method was added to `java.io.BufferedReader` as well:

```
Stream<String> lines()
```

The stream is lazy and its elements are the lines read from the `BufferedReader` .

Key Points

- In Java 8, new methods that return implementations of the `Stream` interface have been added to `java.nio.file.Files` .
- The returned streams are **LAZY**, which means that the elements are not loaded (or read) until they are used.
- The use of a `try-with-resources` with these methods is recommended so that the stream's `close` method can be invoked to close the file system resources.
- `Files.list()` iterates over a directory to return a stream whose elements are `Path` objects that represent the entries of that directory.
- This method lists directories and files of the specified directory. However, it is not recursive, in other words, it **DOESN'T** traverse subdirectories.
- `Files.walk()` also iterates over a directory in a depth-first strategy to return a stream whose elements are `Path` objects that represent the entries of that directory.
- The difference with `Files.list()` is that `Files.walk()` **DOES** recursively traverse the subdirectories. You can also pass the maximum traversal depth and an option to follow symbolic links.
- `Files.find()` is similar to `Files.walk()` , but takes an additional argument of type `BiPredicate<Path,BasicFileAttributes>` that is used to filter the files and directories.
- `Files.lines()` reads all the lines of a file as a stream of Strings without loading them all into memory.

Self Test

1. Given the following structure and class:

```
/temp/
   /dir1/
      1.txt
   0.txt
```

```
public class Question_25_1 {
    public static void main(String[] args) {
        try(Stream<Path> stream =
                Files.walk(Paths.get("/temp"), 0)) {
            stream.forEach(System.out::println);
        } catch(IOException e) { }
    }
}
```

What is the result?
A. `/temp`
B. `/temp/dir1`
`/temp/0.txt`
C. `/temp/0.txt`
D. Nothing is printed

2. Which of the following statements is true?
A. `Files.find()` has a default subdirectory depth of `Integer.MAX_VALUE` .
B. `Files.find()` follows symbolic links by default.
C. `Files.walk()` follows symbolic links by default.
D. `Files.walk()` traverses subdirectories recursively.

3. Which of the following options is equivalent to

```
Files.walk(Paths.get("."))
    .filter(p -> p.toString().endsWith("txt"));
```

A.

```
Files.list(Paths.get("."))
    .filter(p -> p.toString().endsWith("txt"));
```

B.

```
Files.find(Paths.get("."),
    (p,a) -> p.toString().endsWith("txt"));
```

C.

```
Files.find(Paths.get("."), Integer.MAX_VALUE,
     p -> p.toString().endsWith("txt"));
```

D.

```
Files.find(Paths.get("."), Integer.MAX_VALUE,
    (p,a) -> p.toString().endsWith("txt"));
```

4. Which is the behavior of `Files.lines(Path)` if the `Path` object represents a file that doesn't exist?
A. It returns an empty stream.
B. It creates the file.
C. It throws an `IOException` when the method is called.
D. It throws an `IOException` when the stream is first used.

Open answers page

Do you have a problem or something to say?

Report an issue with the book

Contact me

24. NIO.2 26. Thread Basics